Distributed Multithreaded Caching $D$ Compiler

Robert Schadek

May 1, 2013
1. Introduction
2. Compiler Modifications
   - Overview of Compiler Phases
   - Lexer Parser Communication
   - Multi Threaded Semantic Analysis
   - Caching
   - Distribution
3. The Lexer Generator dex
4. The Parser Generator dalr
5. Conclusion
• basic compiler structure has not really changed since Grace Hopper
• hardware capabilities have improved enormously
• basic compiler structure has not really changed since Grace Hopper
• hardware capabilities have improved enormously

• adapt compiler to changed hardware
• learn everything that might be of interest from container to printf style formatting
• graduate
Ideas

- multithreading: use all CPUs
- caching: use the available RAM
- distributing: distribute work in a network
Ideas

• multithreading: use all CPUs
• caching: use the available RAM
• distributing: distribute work in a network

• lexer generator
• parser generator
• library with container etc.
Overview of Compiler Phases

- lexer
- buffer
- parser
- ast and symbols
- semantic analysis 1
- semantic analysis 2
- semantic analysis 3
- semantic analysis n
- code generator

Logic Phase

Process

data structure

thread
• classic Producer Consumer Problem
• historically a parser asks a lexer for a token
• classic Producer Consumer Problem
• historically a parser asks a lexer for a token

• using IO devices interruptively
  ▶ wastes IO performance
  ▶ OS might move HDD head away
• classic Producer Consumer Problem
• historically a parser asks a lexer for a token

• using IO devices interruptively
  ▶ wastes IO performance
  ▶ OS might move HDD head away

• lexer creates token in a separate thread
• synchronisation is limited by copying multiple tokens at a time
Multi Threaded Semantic Analysis

- semantic analysis checks if the program follow the rules
- this is done by traversing the Abstract Syntax Tree (AST) and looking into the symbol table
Multi Threaded Semantic Analysis

- semantic analysis checks if the program follow the rules
- this is done by traversing the Abstract Syntax Tree (AST) and looking into the symbol table
- analysis should not modify any data
- tests are independent of each other
Multi Threaded Semantic Analysis

- semantic analysis checks if the program follow the rules
- this is done by traversing the Abstract Syntax Tree (AST) and looking into the symbol table

- analysis should not modify any data
- tests are independent of each other

- write test as independent functions
- run tests in parallel
Multi Threaded Semantic Analysis

- semantic analysis checks if the program follow the rules
- this is done by traversing the Abstract Syntax Tree (AST) and looking into the symbol table

- analysis should not modify any data
- tests are independent of each other

- write test as independent functions
- run tests in parallel (without any locking)
Multi Threaded Semantic Analysis Benchmark

Robert Schadek
A Distributed Multithreaded Caching D Compiler
• source files are not independent of each other
• many files get imported many times (e.g. stdio)
• unchanged files do not need to be read from the disk again
• use cached data for distributing work
Caching

• source files are not independent of each other
• many files get imported many times (e.g. stdio)
• unchanged files do not need to be read from the disk again
• use cached data for distributing work

• file level
• token level
• AST level
source files are not independent of each other
many files get imported many times (e.g. stdio)
unchanged files do not need to be read from the disk again
use cached data for distributing work

- file level
- token level
- AST level (here it gets interesting)
Linear Trees

• simplify storing of ASTs in cache
• simplify serializing ASTs
• flattening complete uniform trees is easy (binary heap, d-ary heap)

Robert Schadek
A Distributed Multithreaded Caching D Compiler
Linear Trees

- simplify storing of ASTs in cache
- simplify serializing ASTs
- flattening complete uniform trees is easy (binary heap, d-ary heap)
- ASTs are neither complete nor uniform
Linear Trees

- simplify storing of ASTs in cache
- simplify serializing ASTs
• simplify storing of ASTs in cache
• simplify serializing ASTs

• flattening complete uniform trees is easy (binary heap, d-ary heap)
• ASTs are neither complete nor uniform
int main() {
    return 1;
}
<table>
<thead>
<tr>
<th># nodes</th>
<th>Class based</th>
<th>Struct based</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^8$</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$2^9$</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$2^{11}$</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$2^{12}$</td>
<td>1.0</td>
<td>1.2</td>
</tr>
<tr>
<td>$2^{13}$</td>
<td>1.7</td>
<td>4.8</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>6.1</td>
<td>11.5</td>
</tr>
<tr>
<td>$2^{15}$</td>
<td>10.7</td>
<td>23.0</td>
</tr>
<tr>
<td>$2^{16}$</td>
<td>27.3</td>
<td>47.9</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>53.4</td>
<td>100.2</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td>109.3</td>
<td>192.7</td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>278.9</td>
<td>403.6</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>1246.6</td>
<td>815.3</td>
</tr>
</tbody>
</table>

Tree building time in msecs.

<table>
<thead>
<tr>
<th># nodes</th>
<th>Class based</th>
<th>Struct based</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^8$</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$2^9$</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$2^{11}$</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$2^{12}$</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$2^{13}$</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>$2^{15}$</td>
<td>0.0</td>
<td>2.0</td>
</tr>
<tr>
<td>$2^{16}$</td>
<td>2.1</td>
<td>7.3</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>6.2</td>
<td>19.1</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td>14.3</td>
<td>38.5</td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>31.3</td>
<td>76.7</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>65.0</td>
<td>154.2</td>
</tr>
</tbody>
</table>

Tree traversal time in msecs.
• while developing, many CPUs are idle
• multiplied by the number of workstations in a department
• networks are fast wrt. the size of a source file
• while developing, many CPUs are idle
• multiplied by the number of workstations in an department
• networks are fast wrt. the size of a source file

• distribute the compilation of source files to workstations in the network
• compiler becomes a daemon
The Lexer Generator dex

- deterministic finite automaton (DFA) tokenizer
- table driven
- user can supply error recovery function
- supports UTF-8

▶ transition table is compressed
The Lexer Generator dex

- deterministic finite automaton (DFA) tokenizer
- table driven
- user can supply error recovery function
- supports UTF-8
  - transition table is compressed
### The Lexer Generator \texttt{dex}

<table>
<thead>
<tr>
<th>state mapping</th>
<th>input mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>row</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
</tr>
</tbody>
</table>

**Original DFA Table**

<table>
<thead>
<tr>
<th>transition table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>state mapping</th>
<th>input mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>row</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>

**Minimized DFA Table**

<table>
<thead>
<tr>
<th>transition table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>
The Parser Generator dalr

- (g|l)lr1 parser generator
- table driven
- accepts all of Chomsky 2 (context free grammars)
The Parser Generator dalr

- (g|a)lr1 parser generator
- table driven
- accepts all of Chomsky 2 (context free grammars)
  - user code required to remove ambiguities
Summary

- table driven unicode Lexer possible but infeasible
- splitting lexer and parser works well
- caching has great potential
  - especially with linear trees
- multi threaded semantic analysis is a good approach
Summary

- table driven unicode Lexer possible but infeasible
- splitting lexer and parser works well
- caching has great potential
  - especially with linear trees
- multi threaded semantic analysis is a good approach
  - if not for speed, then at least for clean code
Using D

Robert Schadek
A Distributed Multithreaded Caching D Compiler

What is already there:

• fast turnaround time (crash-fix-build-run)

What will be there:

• good documentation

• : instead of ..

• containers
What is already there:

- fast turnaround time (crash-fix-build-run)
- compact expressive code
What is already there:

- fast turnaround time (crash-fix-build-run)
- compact expressive code

What will be there:

- good documentation
- : instead of ..
- containers
The most dangerous phrase in the language is, “We’ve always done it this way.”

Rear Admiral Grace Murray Hopper  
(December 9, 1906 – January 1, 1992)

https://github.com/burner/libhurt  
https://github.com/burner/dex  
https://github.com/burner/dalr  
https://github.com/burner/dmcd  
http://www.svs.informatik.uni-oldenburg.de/60865.html